# Introduction to Parallel Programming

## Part 2: Advanced Concepts

*Argonne National Laboratory*

# Presentation Plan

- Advanced MPI Topics
  - Parallel I/O
  - One sided communication
- Brief introduction to PETSc library with a CFD example run on thousands of processors

# MPI-1

- MPI is a message-passing library interface standard.
  - Specification, not implementation
  - Library, not a language
  - Classical message-passing programming model
- MPI was defined (1994) by a broadly-based group of parallel computer vendors, computer scientists, and applications developers.
  - 2-year intensive process
- Implementations appeared quickly and now MPI is taken for granted as vendor-supported software on any parallel machine.
- Free, portable implementations exist for clusters (MPICH, LAM, OpenMPI) and other environments (MPICH)

# MPI-2

- Same process of definition by MPI Forum
- MPI-2 is an extension of MPI
  - Extends the message-passing *model*.
    - Parallel I/O
    - Remote memory operations (one-sided)
    - Dynamic process management
  - Adds other functionality
    - C++ and Fortran 90 bindings
      - similar to original C and Fortran-77 bindings
    - Language interoperability
    - MPI interaction with threads

# MPI-2 Implementation Status

- Most parallel computer vendors now support MPI-2 on their machines
  - Except in some cases for the dynamic process management functions, which require interaction with other system software
- Cluster MPIs, such as MPICH2 and LAM, support most of MPI-2 including dynamic process management

# Parallel I/O

# What does Parallel I/O Mean?

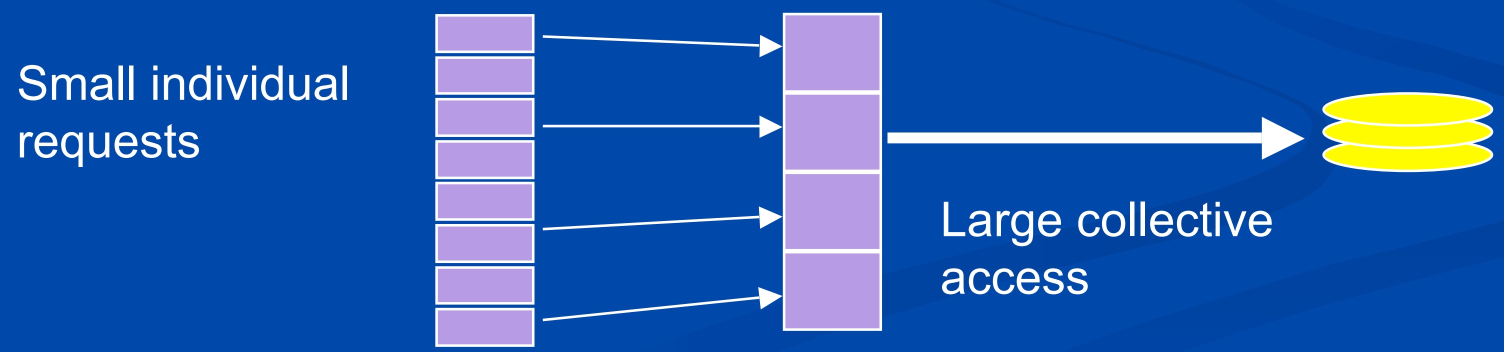
- At the program level:
  - Concurrent reads or writes from multiple processes to a common file
- At the system level:
  - A parallel file system and hardware that support such concurrent access

# Why MPI is a Good Setting for Parallel I/O

- Writing is like sending and reading is like receiving.
- Any parallel I/O system will need:
  - collective operations
  - user-defined datatypes to describe both memory and file layout
  - communicators to separate application-level message passing from I/O-related message passing
  - non-blocking operations
- lots of MPI-like machinery

# Collective I/O and MPI

- A critical optimization in parallel I/O
- All processes (in the communicator) must call the collective I/O function
- Allows communication of "big picture" to file system
  - Framework for I/O optimizations at the MPI-IO layer
- Basic idea: build large blocks, so that reads/writes in I/O system will be large
  - Requests from different processes may be merged together
  - Particularly effective when the accesses of different processes are noncontiguous and interleaved



# Collective I/O Functions

- **`MPI_File_write_at_all`**, etc.
  - **`_all`** indicates that all processes in the group specified by the communicator passed to **`MPI_File_open`** will call this function
  - **`_at`** indicates that the position in the file is specified as part of the call; this provides thread-safety and clearer code than using a separate "seek" call
- Each process specifies only its own access information — the argument list is the same as for the non-collective functions

# The Other Collective I/O Calls

- `MPI_File_seek`
- `MPI_File_read_all`
- `MPI_File_write_all`
- `MPI_File_read_at_all`
- `MPI_File_write_at_all`
- `MPI_File_read_ordered`
- `MPI_File_write_ordered`

like Unix I/O

combine seek and I/O
for thread safety

use shared file pointer

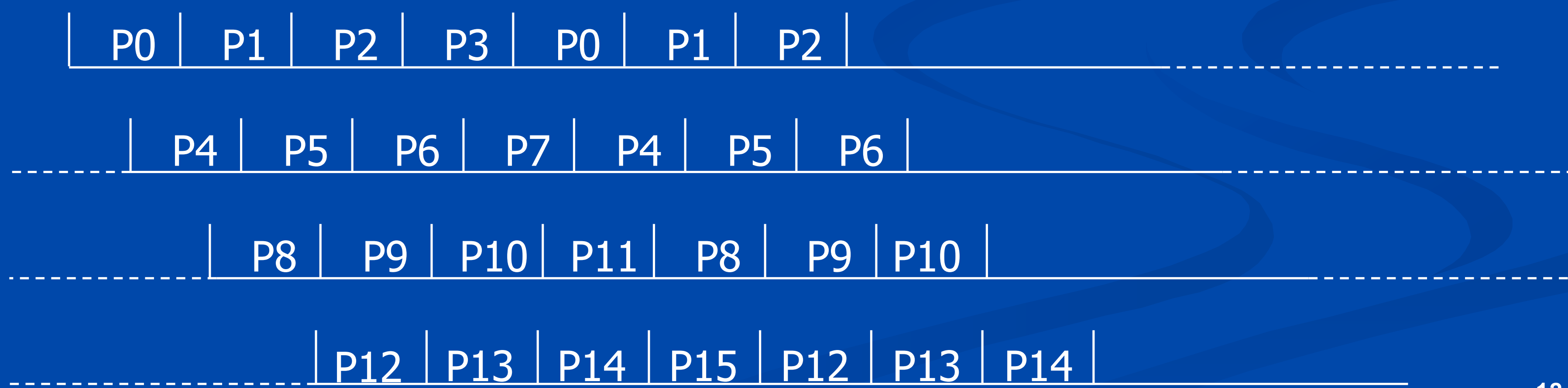
# Example: Distributed Array Access

Large array distributed among 16 processes

| | | | |
|---|---|---|---|
| P0 | P1 | P2 | P3 |
| P4 | P5 | P6 | P7 |
| P8 | P9 | P10 | P11 |
| P12 | P13 | P14 | P15 |

Each square represents a subarray in the memory of a single process

Access Pattern in the file



# Level-0 Access

- Each process makes one independent read request for each row in the local array (as in Unix)

```
call MPI_File_open(..., file, ...,fh,ierr)
do i=1, n_local_rows
    call MPI_File_seek(fh, ..., ierr)
    call MPI_File_read(fh, a(i,0),...,ierr)
enddo
call MPI_File_close(fh, ierr)
```

# Level-1 Access

- Similar to level 0, but each process uses collective I/O functions

```
call MPI_File_open(MPI_COMM_WORLD, file,&
                  ..., fh, ierr)
do i=1,n_local_rows
    call MPI_File_seek(fh, ..., ierr)
    call MPI_File_read_all(fh, a(i,0), ...,&
                          ierr)
enddo
call MPI_File_close(fh,ierr)
```

# Level-2 Access

- Each process creates a derived datatype to describe the noncontiguous access pattern, defines a file view, and calls independent I/O functions

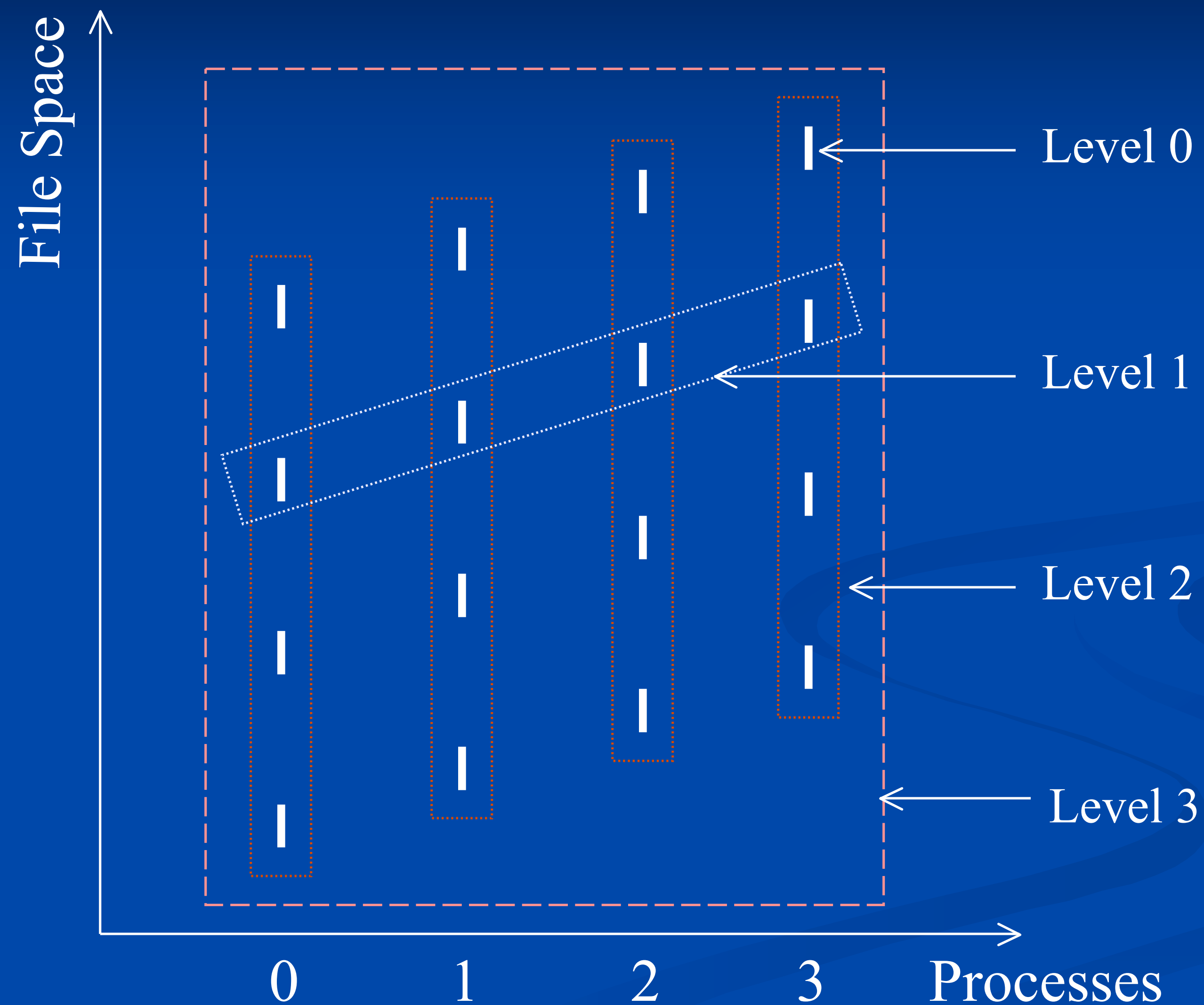
```
call MPI_Type_create_subarray(..., &
                              subarray, ..., ierr)
call MPI_Type_commit(subarray, ierr)
call MPI_File_open(..., file,..., fh, ierr)
call MPI_File_set_view(fh, ..., subarray,&
         ..., ierr)
call MPI_File_read(fh, A, ..., ierr)
call MPI_File_close(fh, ierr )
```

# Level-3 Access

- Similar to level 2, except that each process uses collective I/O functions

```
call MPI_Type_create_subarray(..., &
                subarray, ierr )
call MPI_Type_commit(subarray, ierr )
call MPI_File_open(MPI_COMM_WORLD, file,&
                ..., fh, ierr )
call MPI_File_set_view(fh, ..., subarray,&
                ..., ierr )
call MPI_File_read_all(fh, A, ..., ierr)
call MPI_File_close(fh,ierr)
```

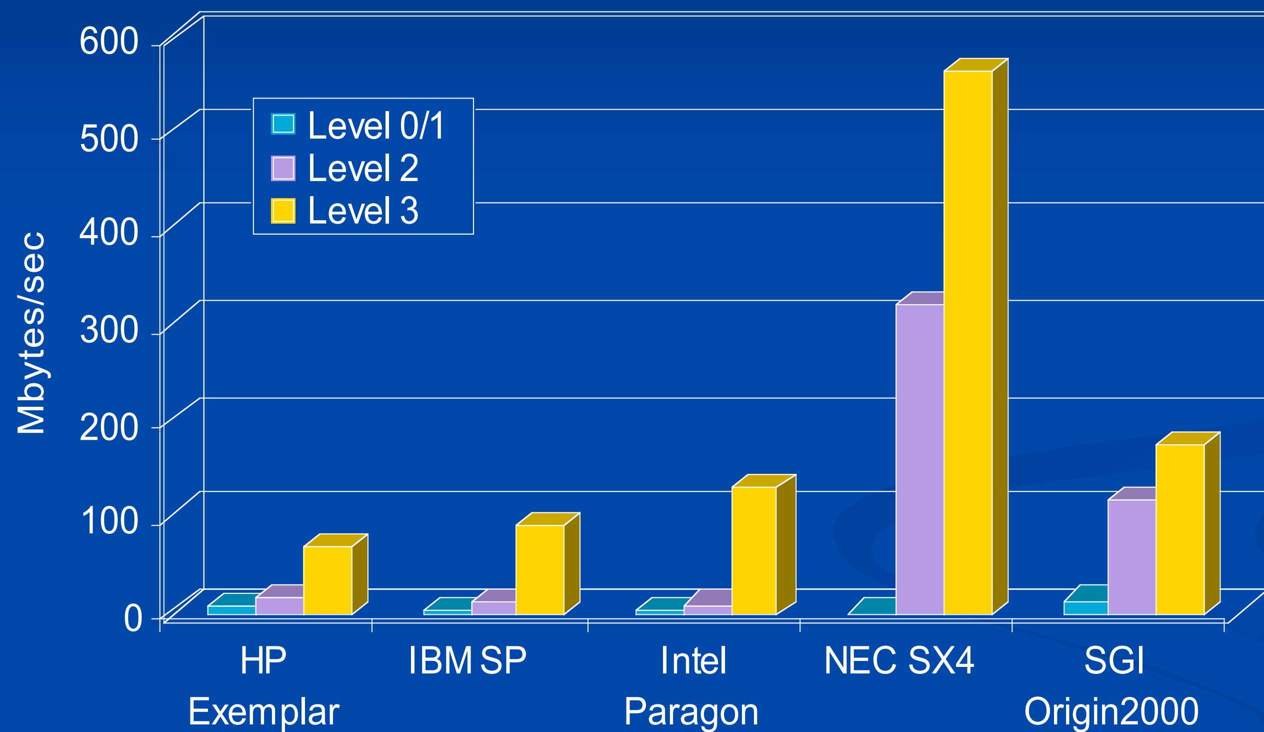
# The Four Levels of Access



# Optimizations

- Given complete access information, an implementation can perform optimizations such as:
  - Data Sieving: Read large chunks and extract what is really needed
  - Collective I/O: Merge requests of different processes into larger requests
  - Improved prefetching and caching

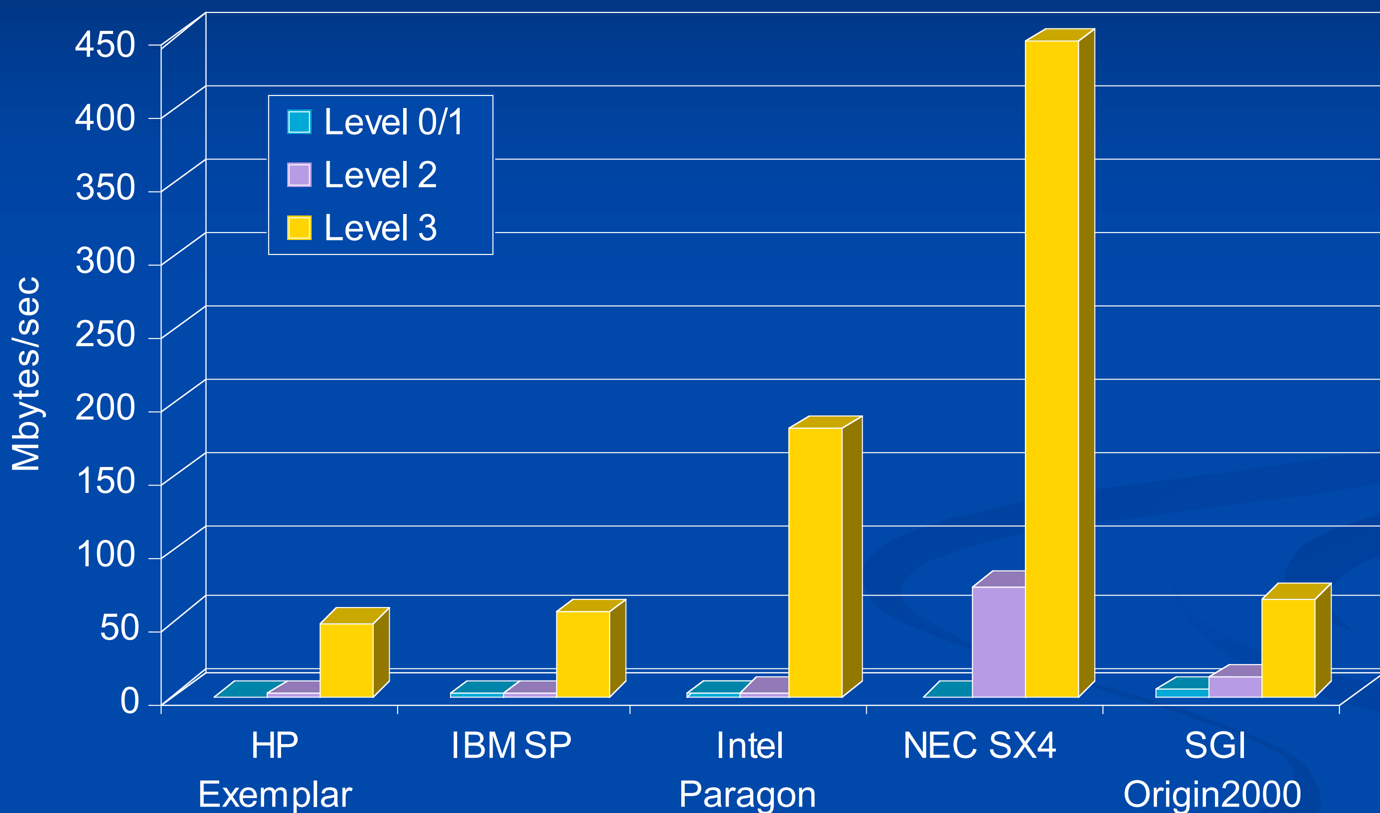
# Distributed Array Access: Read Bandwidth




64 procs | 64 procs | 256 procs | 8 procs | 32 procs

Array size: 512 x 512 x 512

# Distributed Array Access: Write Bandwidth




64 procs | 64 procs | 256 procs | 8 procs | 32 procs

Array size: 512 x 512 x 512

# Portable File Formats

- Ad-hoc file formats
  - Difficult to collaborate
  - Cannot leverage post-processing tools
- MPI provides external32 data encoding
- High level I/O libraries
  - netCDF and HDF5
  - Better solutions than external32
    - Define a "container" for data
      - Describes contents
      - May be queried (self-describing)
    - Standard format for metadata about the file
    - Wide range of post-processing tools available

# File Interoperability in MPI-IO

- Users can optionally create files with a portable binary data representation
- "datarep" parameter to **`MPI_File_set_view`**
- **`native`** - default, same as in memory, not portable
- **`external32`** - a specific representation defined in MPI, (basically 32-bit big-endian IEEE format), portable across machines and MPI implementations
- **`internal`** – implementation-defined representation providing an implementation-defined level of portability
  - Not used by anyone we know of…

# Higher Level I/O Libraries

- Scientific applications work with structured data and desire more self-describing file formats
- netCDF and HDF5 are two popular “higher level” I/O libraries
  - Abstract away details of file layout
  - Provide standard, portable file formats
  - Include metadata describing contents
- For parallel machines, these should be built on top of MPI-IO
  - HDF5 has an MPI-IO option
    - http://hdf.ncsa.uiuc.edu/HDF5/

# Parallel netCDF (PnetCDF)

- (Serial) netCDF
  - API for accessing multi-dimensional data sets
  - Portable file format
  - Popular in both fusion and climate communities
- Parallel netCDF
  - Very similar API to netCDF
  - Tuned for better performance in today's computing environments
  - Retains the file format so netCDF and PnetCDF applications can share files
  - PnetCDF builds on top of any MPI-IO implementation

Cluster

| |
|---|
| PnetCDF |
| ROMIO |
| PVFS2 |

IBM SP

| |
|---|
| PnetCDF |
| IBM MPI |
| GPFS |

# Exchanging Data with RMA

# Remote Memory Access in MPI-2 (also called One-Sided Operations)

- Goals of MPI-2 RMA Design
  - Balancing efficiency and portability across a wide class of architectures
    - shared-memory multiprocessors
    - NUMA architectures
    - distributed-memory MPP's, clusters
    - Workstation networks
  - Retaining "look and feel" of MPI-1
  - Dealing with subtle memory behavior issues: cache coherence, sequential consistency

# Mesh Communication

- Recall how we designed the parallel implementation
  - Determine source and destination data
- Do not need full generality of send/receive
  - Each process can completely define what data needs to be moved to itself, relative to each processes local mesh
    - Each process can "get" data from its neighbors
  - Alternately, each can define what data is needed by the neighbor processes
    - Each process can "put" data to its neighbors

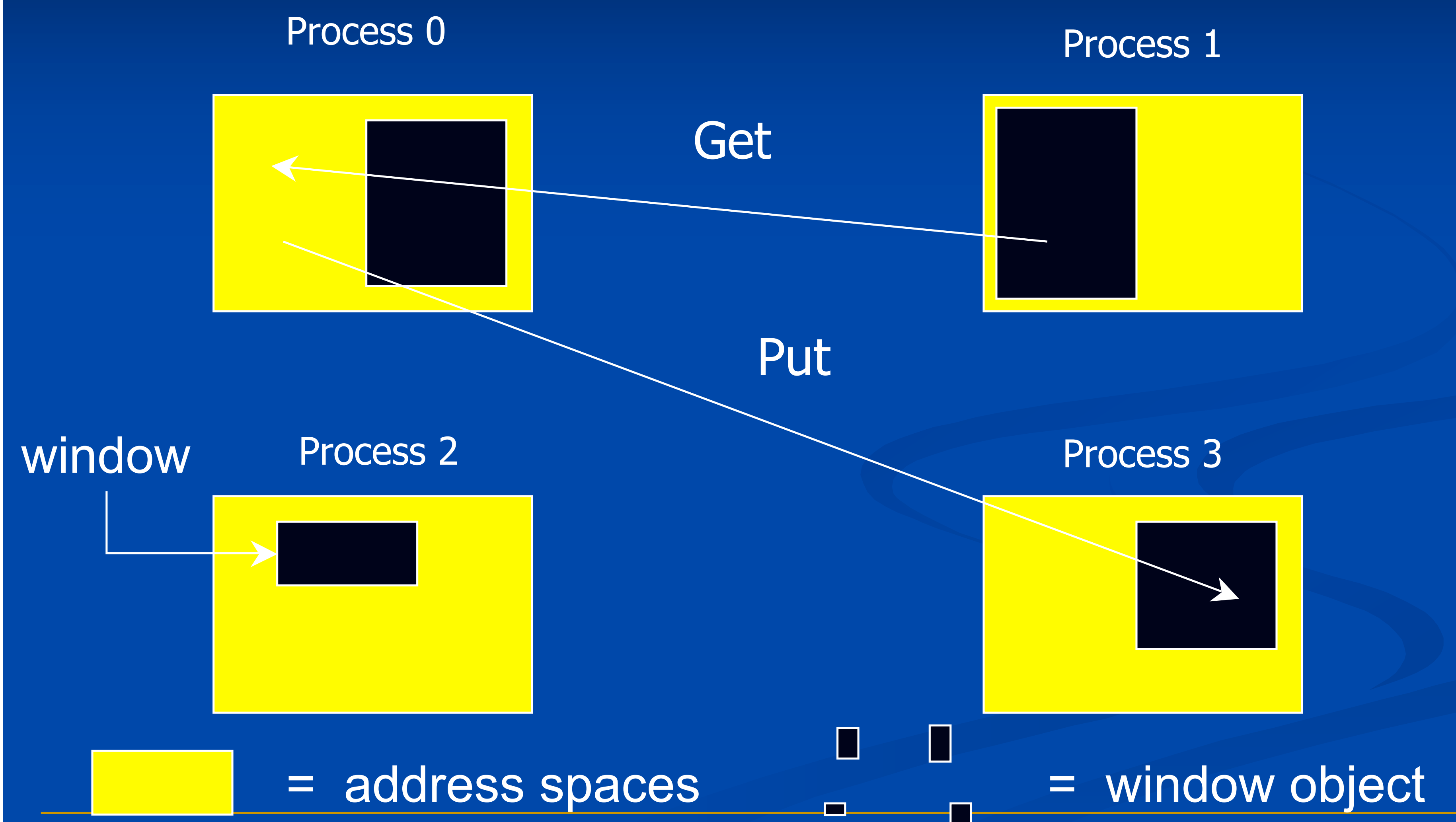
# Remote Memory Access

- Separates data transfer from indication of completion (synchronization)
- In message-passing, they are combined

| Proc 0 | Proc 1 |
|---|---|
| store | |
| send | receive |
| | load |

| Proc 0 | Proc 1 |
|---|---|
| fence | fence |
| put | |
| fence | fence |
| | load |
| ***or*** | |
| store | |
| fence | fence |
| | get |

# Remote Memory Access Windows and Window Objects



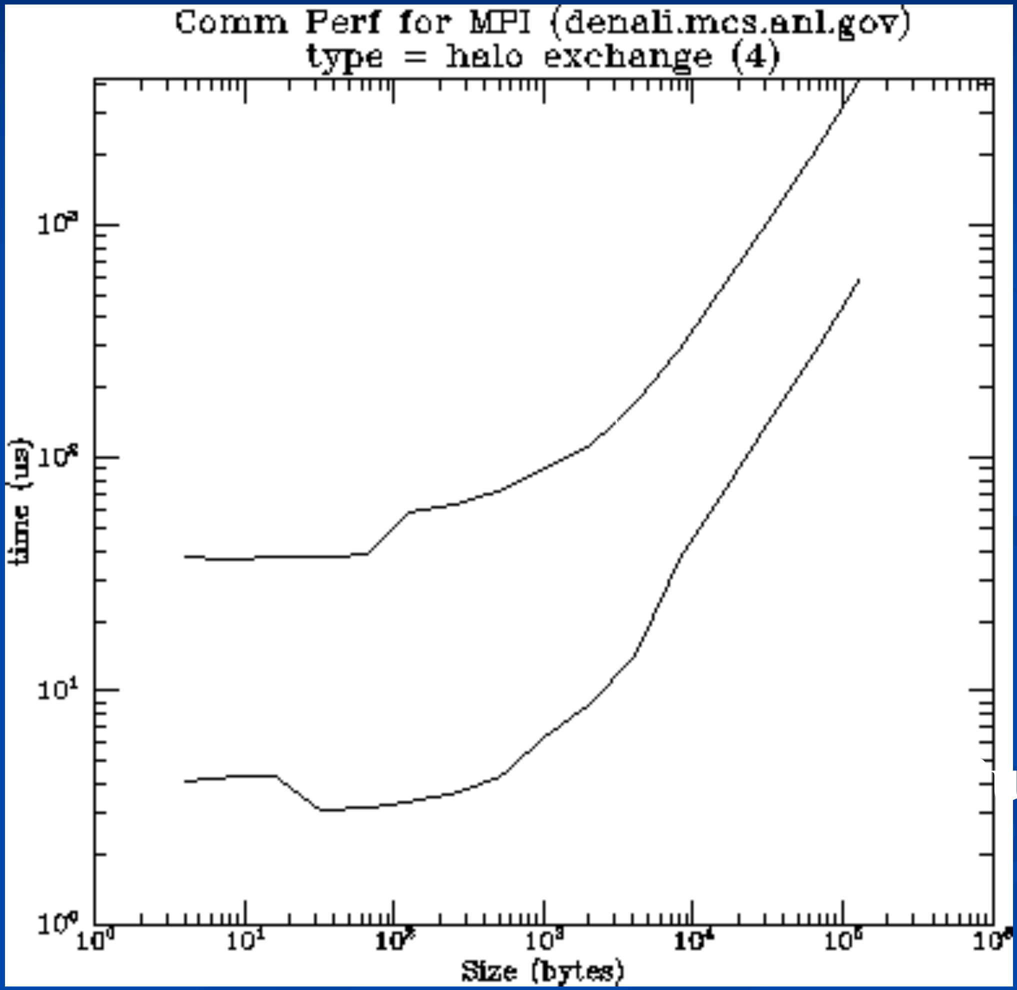
# Basic RMA Functions for Communication

- `MPI_Win_create` exposes local memory to RMA operation by other processes in a communicator
  - Collective operation
  - Creates window object
- `MPI_Win_free` deallocates window object

- `MPI_Put` moves data from local memory to remote memory
- `MPI_Get` retrieves data from remote memory into local memory
- `MPI_Accumulate` updates remote memory using local values
- Data movement operations are non-blocking
- **Subsequent synchronization on window object needed to ensure operation is complete**

# Send vs. Put

- MPI_Put can be much faster that MPI Point-to-point
  - 4 neighbor exchange on SGI Origin




ut

# Advantages of RMA Operations

- Can do multiple data transfers with a single synchronization operation
- Some irregular communication patterns can be more economically expressed
- Can be significantly faster than send/receive on systems with hardware support for remote memory access, such as shared memory systems

# Irregular Communication Patterns with RMA

- If communication pattern is not known *a priori*, the send-recv model requires an extra step to determine how many sends-recvs to issue
- RMA, however, can handle it easily because only the origin or target process needs to issue the put or get call
- This makes dynamic communication easier to code in RMA

# RMA Window Objects

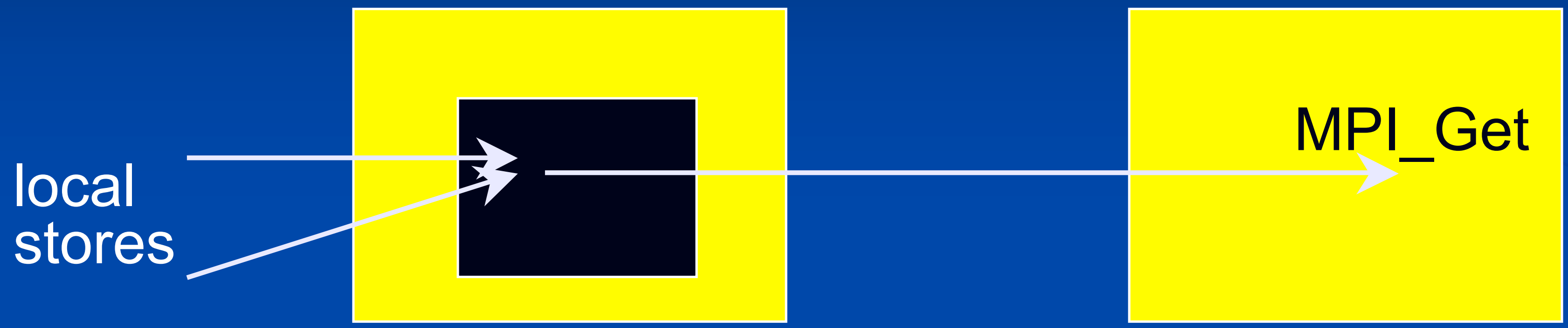
```
MPI_Win_create(base, size, disp_unit,
  info,
                          comm, win)
```

- Exposes memory given by **(base, size)** to RMA operations by other processes in **comm**
- **win** is window object used in RMA operations
- **disp_unit** scales displacements:
  - 1 (no scaling) or **sizeof(type)**, where window is an array of elements of type **type**
  - Allows use of array indices
  - Allows heterogeneity

# RMA Communication Calls

- **`MPI_Put`** - stores into remote memory
- **`MPI_Get`** - reads from remote memory
- **`MPI_Accumulate`** - updates remote memory
- All are non-blocking: data transfer is described, maybe even initiated, but may continue after call returns
- Subsequent synchronization on window object is needed to ensure operations are complete

# The Synchronization Issue




- Issue: Which value is retrieved?
  - Some form of synchronization is required between local load/stores and remote get/put/accumulates
- MPI provides multiple forms

# Synchronization with Fence

Simplest methods for synchronizing on window objects:

- **`MPI_Win_fence`** - like barrier

| Process 0 | Process 1 |
| --- | --- |
| MPI_Win_fence(win) | MPI_Win_fence(win) |
| MPI_Put | |
| MPI_Put | |
| MPI_Win_fence(win) | MPI_Win_fence(win) |

# PETSc
# Portable Extensible Toolkit for Scientific Computing

**http://www.mcs.anl.gov/petsc**

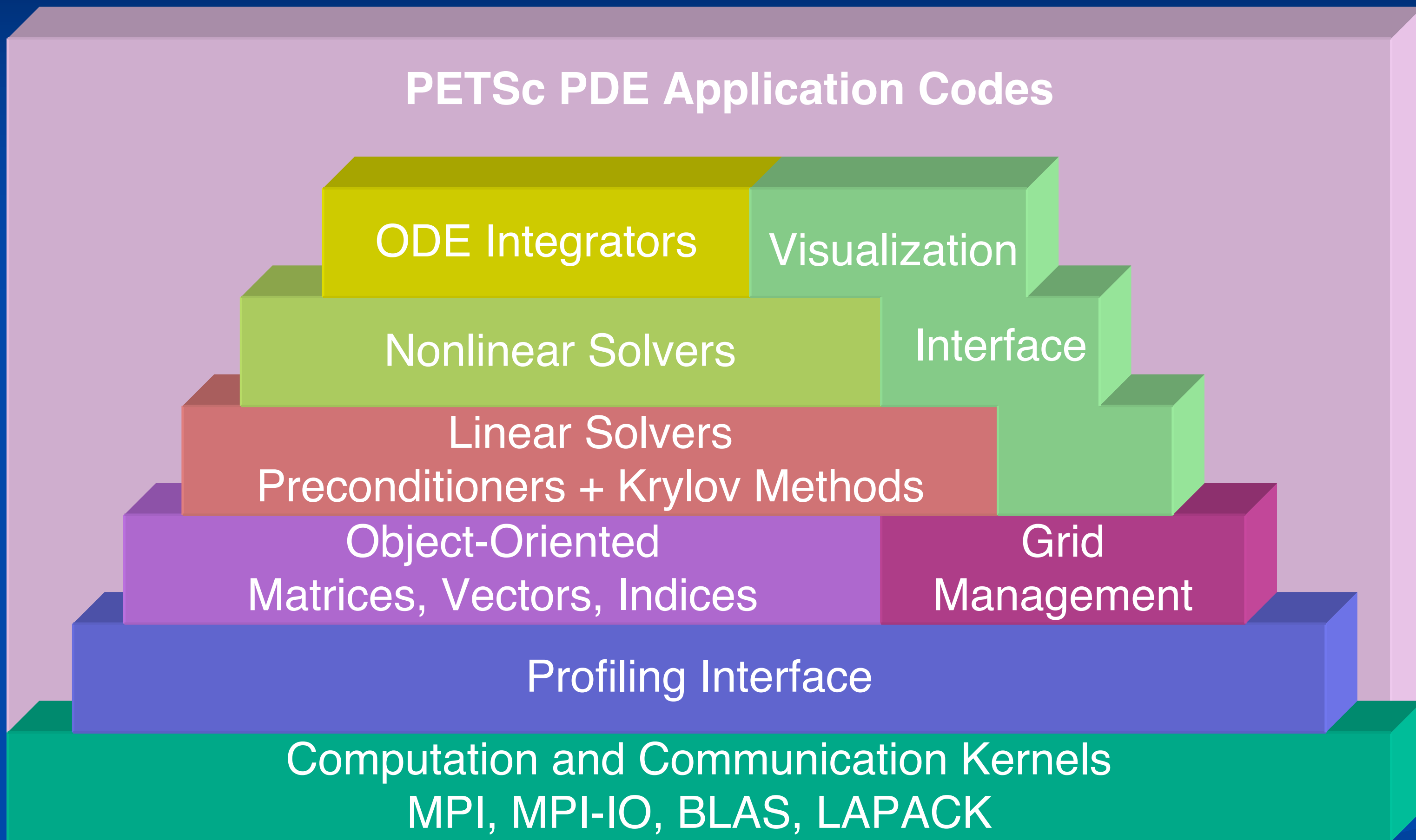
# The Role of PETSc

- Developing parallel, non-trivial PDE solvers that deliver high performance is still difficult and requires months (or even years) of concentrated effort.
- PETSc is a toolkit that can ease these difficulties and reduce the development time, but it is not a black-box PDE solver nor a silver bullet.

# Overview of PETSc (http://www.mcs.anl.gov/petsc)

- Gives relatively high-level expression to preconditioned iterative linear solvers, and Newton iterative methods
- Ports wherever MPI ports; committed to progressive MPI tuning
- Permits great flexibility (through object-oriented philosophy) for algorithmic innovation
- Callable from FORTRAN77, C, and C++

# Structure of PETSc

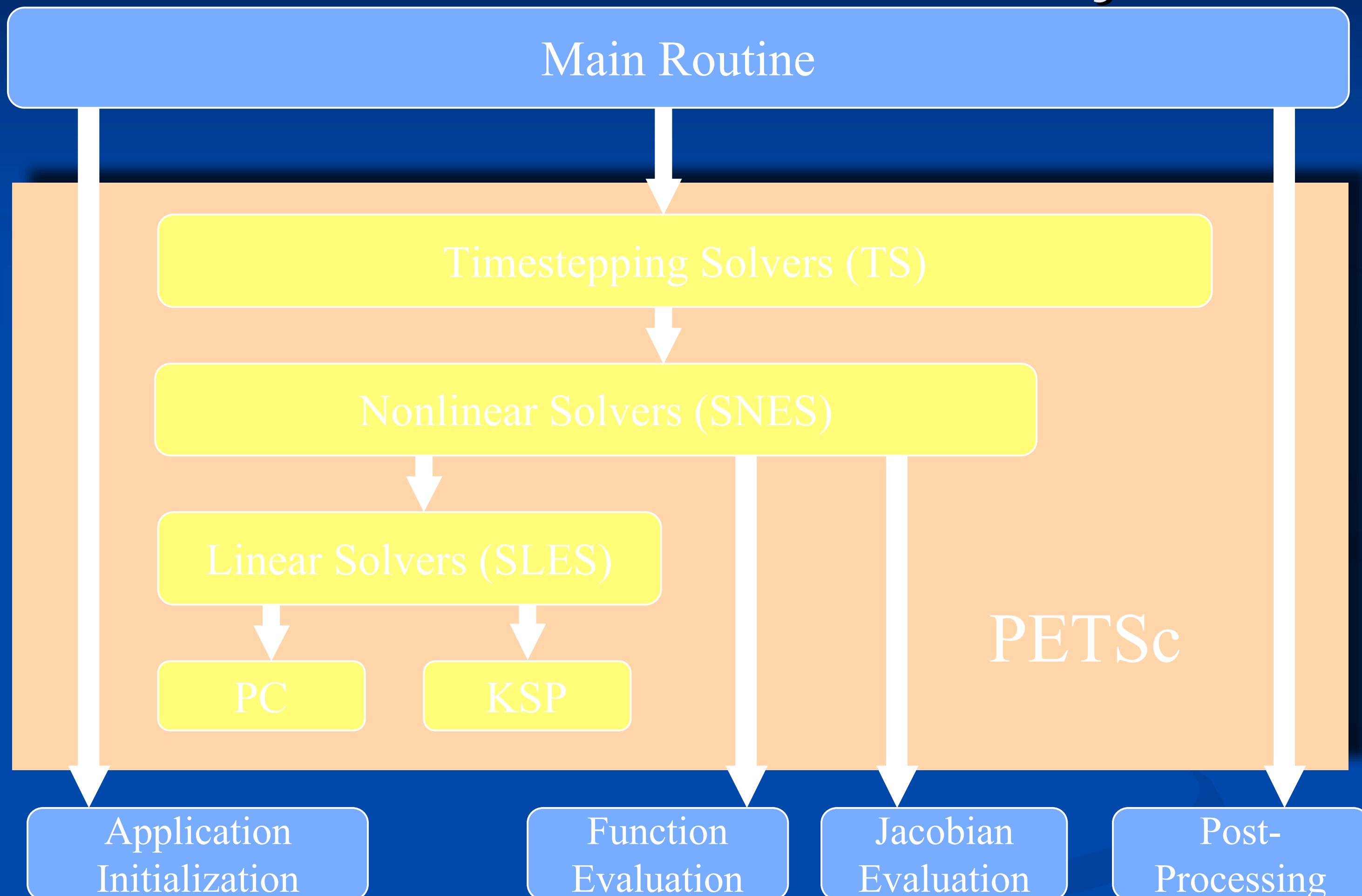


# What is not in PETSc?

- Higher level representations of PDEs
  - Unstructured mesh generation and manipulation
  - Discretizations
- Load balancing
- Sophisticated visualization capabilities
- Optimization and sensitivity

But PETSc does interface to external software that provides some of this functionality.

# Flow of Control: User Code/PETSc Library



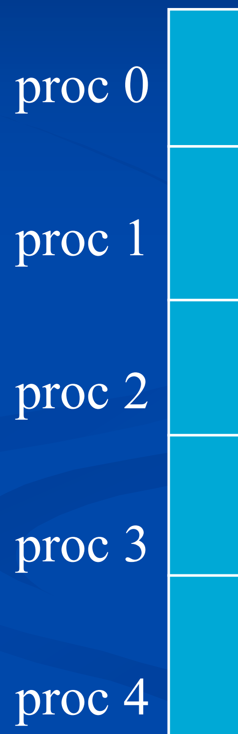

User code    PETSc code

# PETSc Objects

- Vectors
  - Sequential and parallel
- Matrices
  - Sequential and parallel
- Linear Solvers
  - ksp, preconditioners
- Nonlinear Solvers
- Time integration

# Vectors

- What are PETSc vectors?
  - Fundamental objects for storing field solutions, right-hand sides, etc.
  - Each process locally owns a subvector of contiguously numbered global indices
- Create vectors via
  - VecCreate(MPI_Comm,Vec *)
    - MPI_Comm - processes that share the vector
  - VecSetSizes( Vec, int, int )
    - number of elements local to this process
    - or total number of elements
  - VecSetType(Vec,VecType)
    - Where VecType is
      - VEC_SEQ, VEC_MPI, or VEC_SHARED
    - VecSetFromOptions(Vec) lets you set the type at *runtime*




data objects: vectors

# Creating a Vector

Use PETSc to get value from command line

```
Vec x;
int n;
…
PetscInitialize(&argc,&argv,(char*)0,help);
PetscOptionsGetInt(PETSC_NULL,"-n",&n,PETSC_NULL);
…
VecCreate(PETSC_COMM_WORLD,&x);
VecSetSizes(x,PETSC_DECIDE,n);
VecSetType(x,VEC_MPI);
VecSetFromOptions(x);
```

Global size

PETSc determines local size

# How Can We Use a PETSc Vector

- PETSc supports “data structure-neutral” objects
  - distributed memory “shared nothing” model
  - single processors and shared memory systems
- PETSc vector is a “handle” to the real vector
  - Allows the vector to be distributed across many processes
  - To access the *elements* of the vector, we cannot simply do

    ```
    for (i=0; i<n; i++) v[i] = i;
    ```
  - We do not *require* that the programmer work only with the “local” part of the vector; we permit operations, such as setting an element of a vector, to be performed globally

# Vector Assembly

- A three step process
  - Each process tells PETSc what values to set or add to a vector component. Once *all* values provided,
  - Begin communication between processes to ensure that values end up where needed
  - (allow other operations, such as some computation, to proceed)
  - Complete the communication
- VecSetValues(Vec,…)
  - number of entries to insert/add
  - indices of entries
  - values to add
  - mode: [INSERT_VALUES,ADD_VALUES]
- VecAssemblyBegin(Vec)
- VecAssemblyEnd(Vec)

# Selected Vector Operations

| Function Name | Operation |
|---|---|
| VecAXPY(Scalar *a, Vec x, Vec y) | $y = y + a*x$ |
| VecAYPX(Scalar *a, Vec x, Vec y) | $y = x + a*y$ |
| VecWAXPY(Scalar *a, Vec x, Vec y, Vec w) | $w = a*x + y$ |
| VecScale(Scalar *a, Vec x) | $x = a*x$ |
| VecCopy(Vec x, Vec y) | $y = x$ |
| VecPointwiseMult(Vec x, Vec y, Vec w) | $w\_i = x\_i *y\_i$ |
| VecMax(Vec x, int *idx, double *r) | $r = max\ x\_i$ |
| VecShift(Scalar *s, Vec x) | $x\_i = s+x\_i$ |
| VecAbs(Vec x) | $x\_i = \|x\_i\|$ |
| VecNorm(Vec x, NormType type , double *r) | $r = \|\|x\|\|$ |

# A Complete PETSc Program

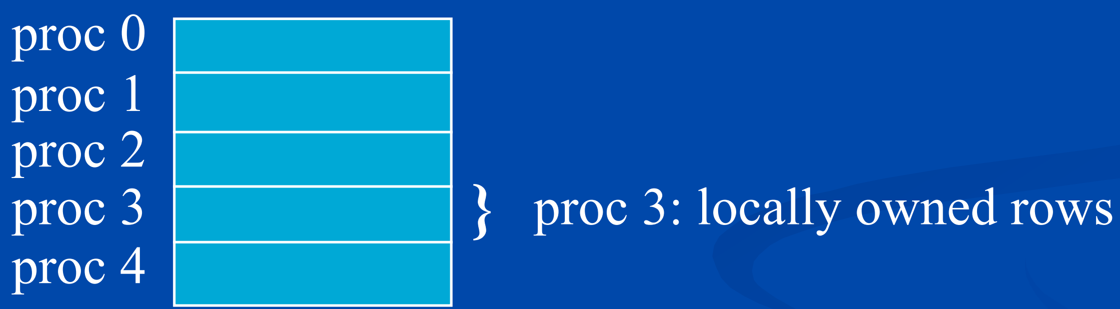
```
#include petscvec.h
int main(int argc,char **argv)
{
  Vec x;
  int n = 20,ierr;
  PetscTruth flg;
  PetscScalar one = 1.0, dot;

  PetscInitialize(&argc,&argv,0,0);
  PetscOptionsGetInt(PETSC_NULL,"-n",&n,PETSC_NULL);
  VecCreate(PETSC_COMM_WORLD,&x);
  VecSetSizes(x,PETSC_DECIDE,n);
  VecSetFromOptions(x);
  VecSet(&one,x);
  VecDot(x,x,&dot);
  PetscPrintf(PETSC_COMM_WORLD,"Vector length %dn",(int)dot);
  VecDestroy(x);
  PetscFinalize();
  return 0;
}
```

# Matrices

- What are PETSc matrices?
  - Fundamental objects for storing linear operators (e.g., Jacobians)
- Create matrices via
  - MatCreate(…,Mat *)
    - MPI_Comm - processes that share the matrix
    - number of localglobal rows and columns
  - MatSetType(Mat,MatType)
    - where MatType is one of
      - default sparse AIJ: MPIAIJ, SEQAIJ
      - block sparse AIJ (for multi-component PDEs): MPIAIJ, SEQAIJ
      - symmetric block sparse AIJ: MPISBAIJ, SAEQSBAIJ
      - block diagonal: MPIBDIAG, SEQBDIAG
      - dense: MPIDENSE, SEQDENSE
      - matrix-free
      - etc.
    - MatSetFromOptions(Mat) lets you set the MatType at *runtime*.

# Parallel Matrix Distribution

Each process locally owns a submatrix of contiguously numbered global rows.




MatGetOwnershipRange(Mat A, int *rstart, int *rend)

- rstart: first locally owned row of global matrix
- rend -1: last locally owned row of global matrix

# Matrix Assembly Example With Parallel Assembly

simple 3-point stencil for 1D discretization

```
Mat     A;
int     column[3], i, start, end,istart,iend;
double value[3];
…
MatCreate(PETSC_COMM_WORLD,
          PETSC_DECIDE,PETSC_DECIDE,n,n,&A);

MatSetFromOptions(A);
MatGetOwnershipRange(A,&start,&end);
/* mesh interior */
istart = start; if (start == 0) istart = 1;
iend = end; if (iend == n-1) iend = n-2;
value[0] = -1.0; value[1] = 2.0; value[2] = -1.0;
for (i=istart; i<iend; i++) {
   column[0] = i-1; column[1] = i; column[2] = i+1;
   MatSetValues(A,1,&i,3,column,value,INSERT_VALUES);
}
/* also must set boundary points  (code for global row 0 and n-1 omitted) */
MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);
```

Choose the global Size of the matrix

Let PETSc decide how to allocate matrix across processes

# Linear Solvers

- Krylov Methods
  - Using PETSc linear algebra, just add:
    - KSPSetOperators(), KSPSetRhs(), KSPSetSolution()
    - KSPSolve()
  - Preconditioners must obey PETSc interface
    - Basically just the KSP interface
  - Can change solver dynamically from the command line

# Nonlinear Solvers

- Using PETSc linear algebra, just add:
  - SNESSetFunction(), SNESSetJacobian()
  - SNESSolve()
- Can access subobjects
  - SNESGetKSP()
  - KSPGetPC()
- Can customize subobjects from the cmd line
  - Could give –sub_pc_type ilu, which would set the subdomain preconditioner to ILU

# Debugging

Support for parallel debugging

- -start_in_debugger  [gdb,dbx,noxterm]
- -on_error_attach_debugger [gb,dbx,noxterm]
- -on_error_abort
- -debugger_nodes 0,1
- -display machinename:0.0

When debugging, it is often useful to place a breakpoint in the function PetscError( ).

# Profiling and Performance Tuning

**Profiling:**

- Integrated profiling using -log_summary
- User-defined events
- Profiling by stages of an application

**Performance Tuning:**

- Matrix optimizations
- Application optimizations
- Algorithmic tuning

# CFD Example: PETSc-FUN3D

- Based on “legacy” (but contemporary) NASA CFD application, with significant F77 code reuse
- Portable, message-passing library-based parallelization, runs on NT boxes through Tflop/s ASCI platforms
- Simple multithreaded extension (for SMP Clusters)
- Sparse, unstructured data, implying memory indirection with only modest reuse
- Wide applicability to other implicitly discretized multiple-scale PDE workloads — of interagency, interdisciplinary interest

# Euler Simulation

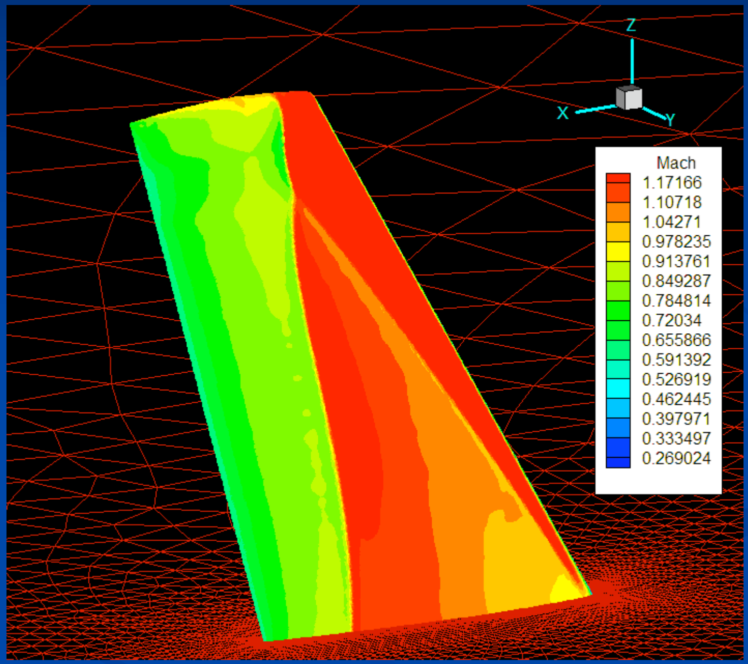
- 3D transonic flow over ONERA M6 wing, at 3.06° angle of attack (exhibits λ-shock at M = 0.839)
- Solve $\frac{\partial Q}{\partial t} + \frac{1}{V}\oint_{\Omega}(\vec{F}\cdot\hat{n})\,d\Omega = 0$

  where

$$Q = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ E \end{bmatrix} \qquad \vec{F}\cdot\hat{n} = \begin{bmatrix} \rho U \\ \rho U u + \hat{n}_x p \\ \rho U v + \hat{n}_y p \\ \rho U w + \hat{n}_z p \\ (E+p)U \end{bmatrix}$$

$$U = \hat{n}_x u + \hat{n}_y v + \hat{n}_z w$$

$$p = (\gamma - 1)\left[E - \rho\frac{\left(u^2 + v^2 + w^2\right)}{2}\right]$$



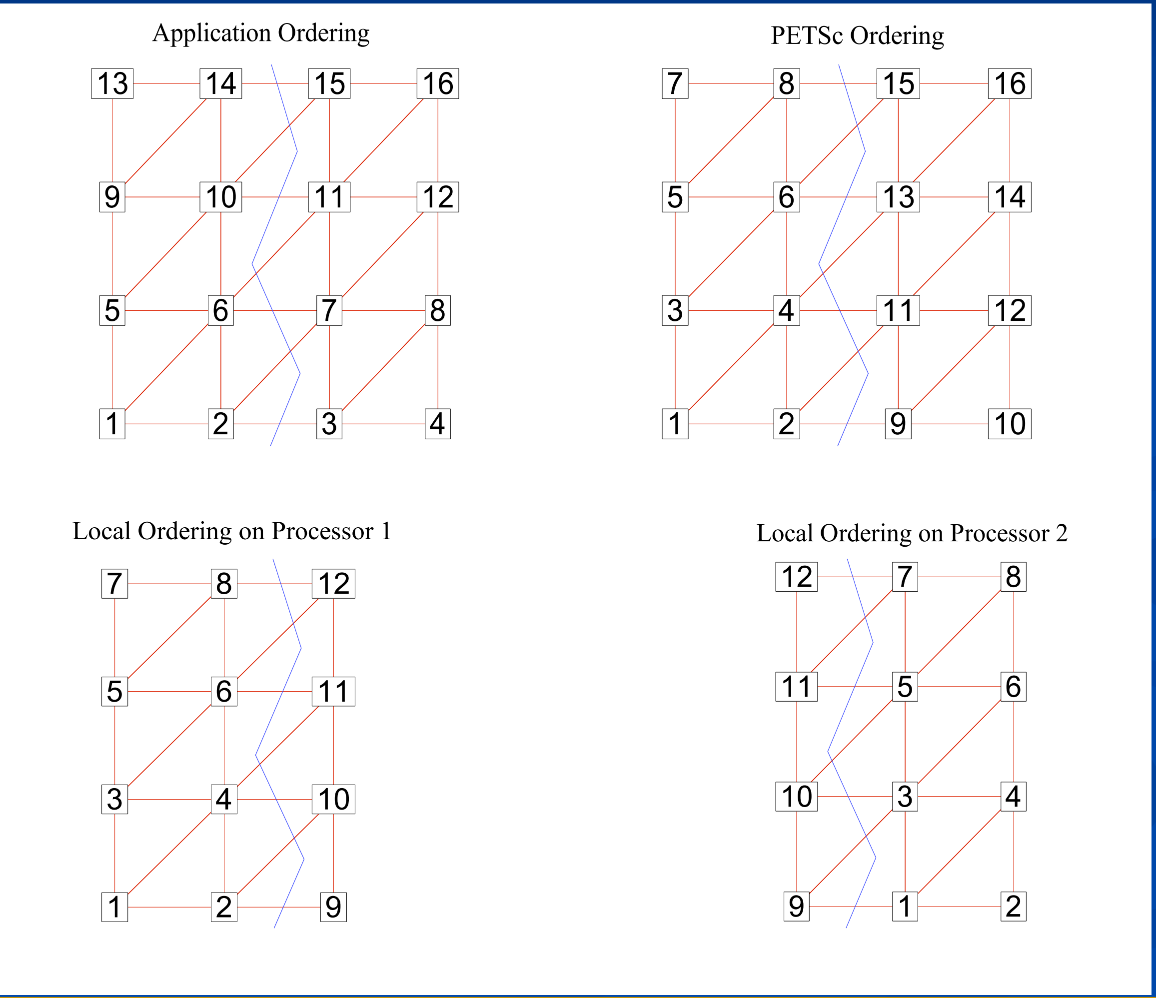

$\rho$ = density, $\boldsymbol{u}$ = velocity, $p$ = pressure

$E$ = energy density

# PETSc-FUN3D Code – Parallelization Approach

- Follow the "owner computes" rule under the dual constraints of minimizing the number of messages and overlapping communication with computation
- Each processor "ghosts" its stencil dependences in its neighbors
- Ghost nodes ordered after contiguous owned nodes
- Domain mapped from (user) global ordering into local orderings
- Scatter/gather operations created between **local sequential** vectors and **global distributed** vectors, based on runtime connectivity patterns

# Different Orderings



# Solving Unstructured Mesh Problems in Serial

- makes them more **memory intensive**
- reduces the **locality in data reference** patterns (which is required to get good cache performance)
- needs high **memory bandwidth** since cache lines might be loaded multiple times
- requires lot of **integer operations** that make these solvers more susceptible to run into **operation issue** limitations

# Solving Unstructured Grid Problems in Parallel: Main Issues

- SPMD parallelization of unstructured grid solvers is complicated by the fact that no two interprocessor data dependency patterns are alike
- The user-provided global ordering may be incompatible with the subdomain-contiguous ordering required for high performance and convenient SPMD coding

# Time-Implicit Newton-Krylov-Schwarz (ΨNKS)

### For nonlinear robustness, NKS iteration is wrapped in time-stepping

```
for (l = 0; l < n_time; l++) {                          # n_time ~ 50
    select time step
    for (k = 0; k < n_Newton; k++) {                         # n_Newton ~ 1
        compute nonlinear residual and Jacobian
         for (j = 0; j < n_Krylov; j++) {                # n_Krylov ~ 60
            forall (i = 0; i < n_Precon ; i++) {
                          solve subdomain problems concurrently
           } // End of loop over subdomains
           perform Jacobian-vector product
           enforce Krylov basis conditions
           update optimal coefficients
           check linear convergence
         } // End of linear solver
         perform DAXPY update
         check nonlinear convergence
     } // End of nonlinear loop
} // End of time-step loop
```

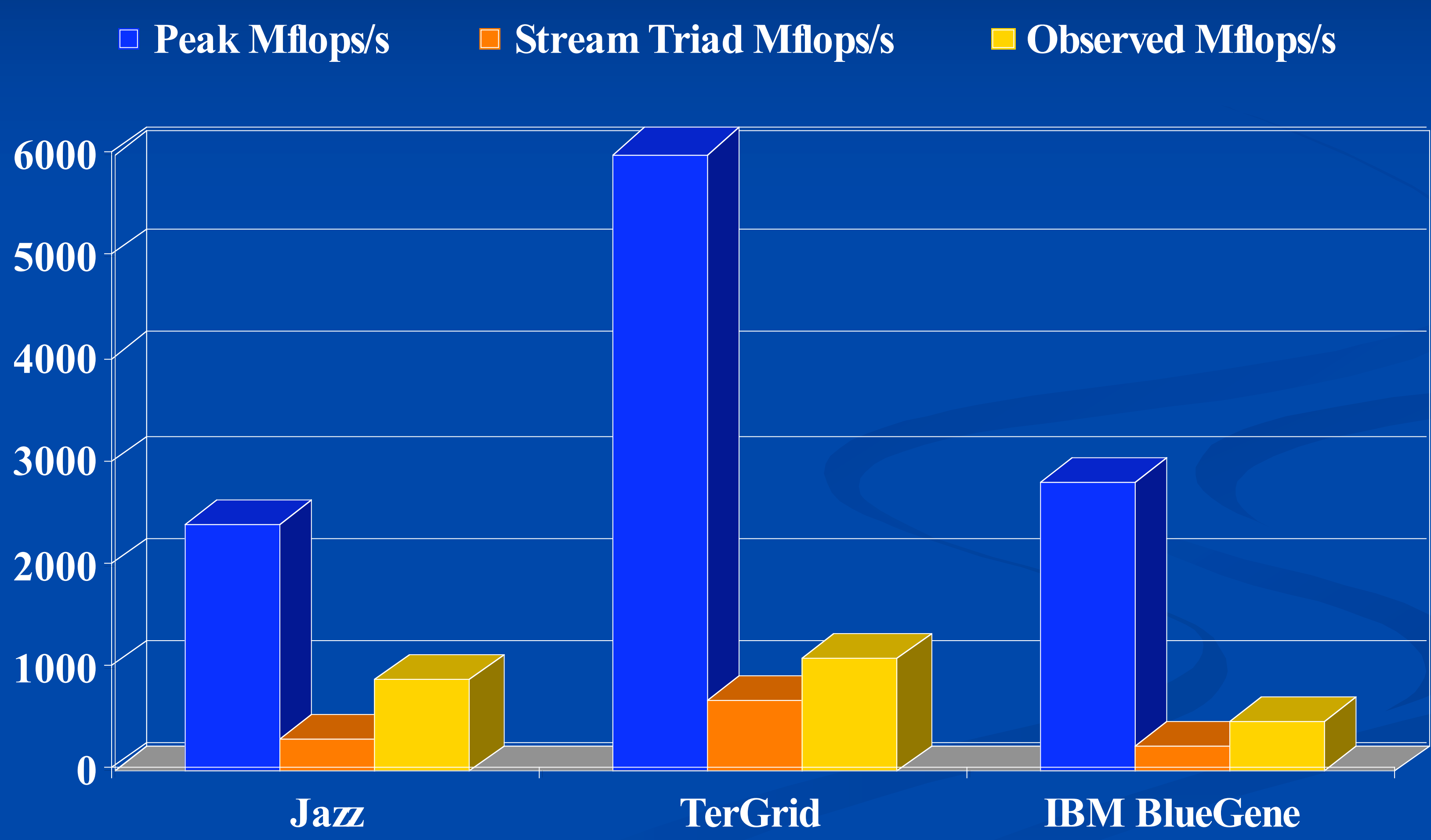
# Primary PDE Solution Kernels

- Vertex-based loops
  - state vector and auxiliary vector updates
- Edge-based “stencil op” loops
  - residual evaluation
  - approximate Jacobian evaluation
  - Jacobian-vector product (often replaced with matrix-free form, involving residual evaluation)
- Sparse, narrow-band recurrences
  - approximate factorization and back substitution
- Vector inner products and norms
  - orthogonalization/conjugation
  - convergence progress and stability checks

# Algorithmic Tuning for NKS Solver

- Continuation parameters: discretization order, initial timestep, timestep evolution
- Newton parameters: convergence tolerance, globalization strategy, Jacobian refresh frequency
- Krylov parameters: convergence tolerance, subspace dimension, restart number, orthogonalization mechanism
- Schwarz parameters: subdomain number, subdomain solver, subdomain overlap, coarse grid usage
- Subproblem parameters: fill level, number of sweeps

# Sequential Performance of PETSc-FUN3D



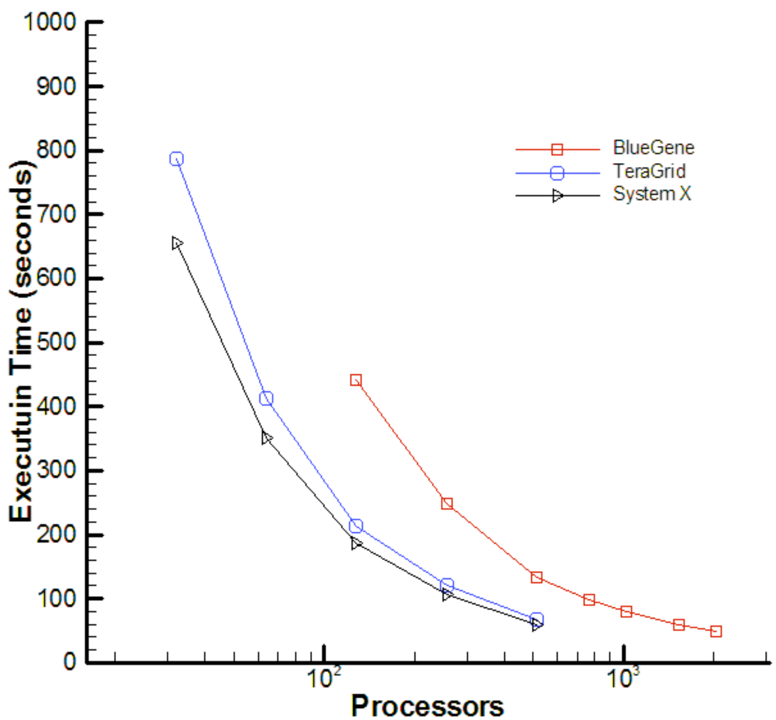
# Parallel Performance of PETSc-FUN3D

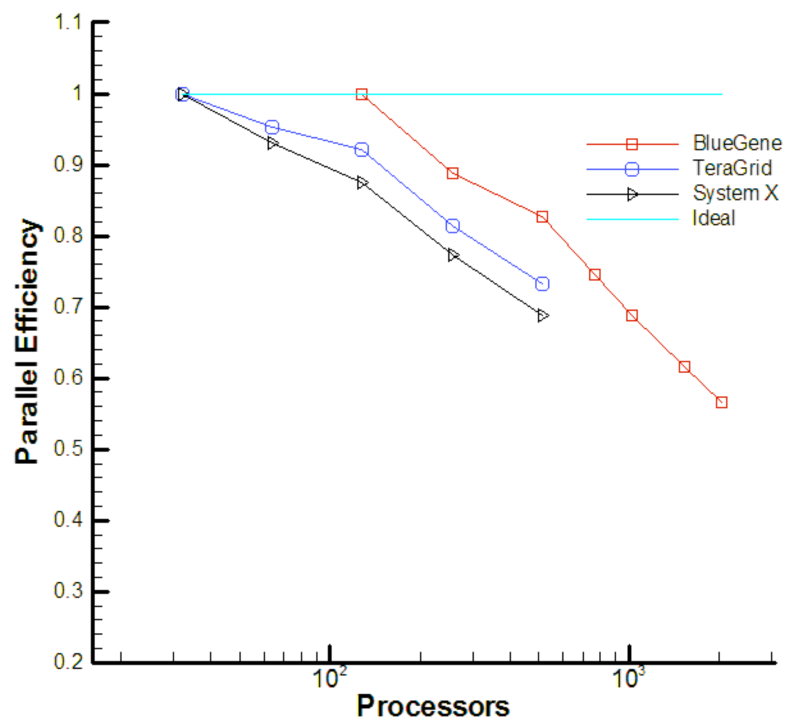
**3D Mesh: 2,761,774 Vertices and 18,945,809 Edges**

**TeraGrid: Dual 1.5 GHz Intel Madison Processors with 4 MB L2 Cache**

**BlueGene: Dual 700 MHz IBM Processors with 4 MB L3 Cache**

**System X: Dual 2.3 GHz PowerPC 970FX processors with 0.5 MB L2 Cache**



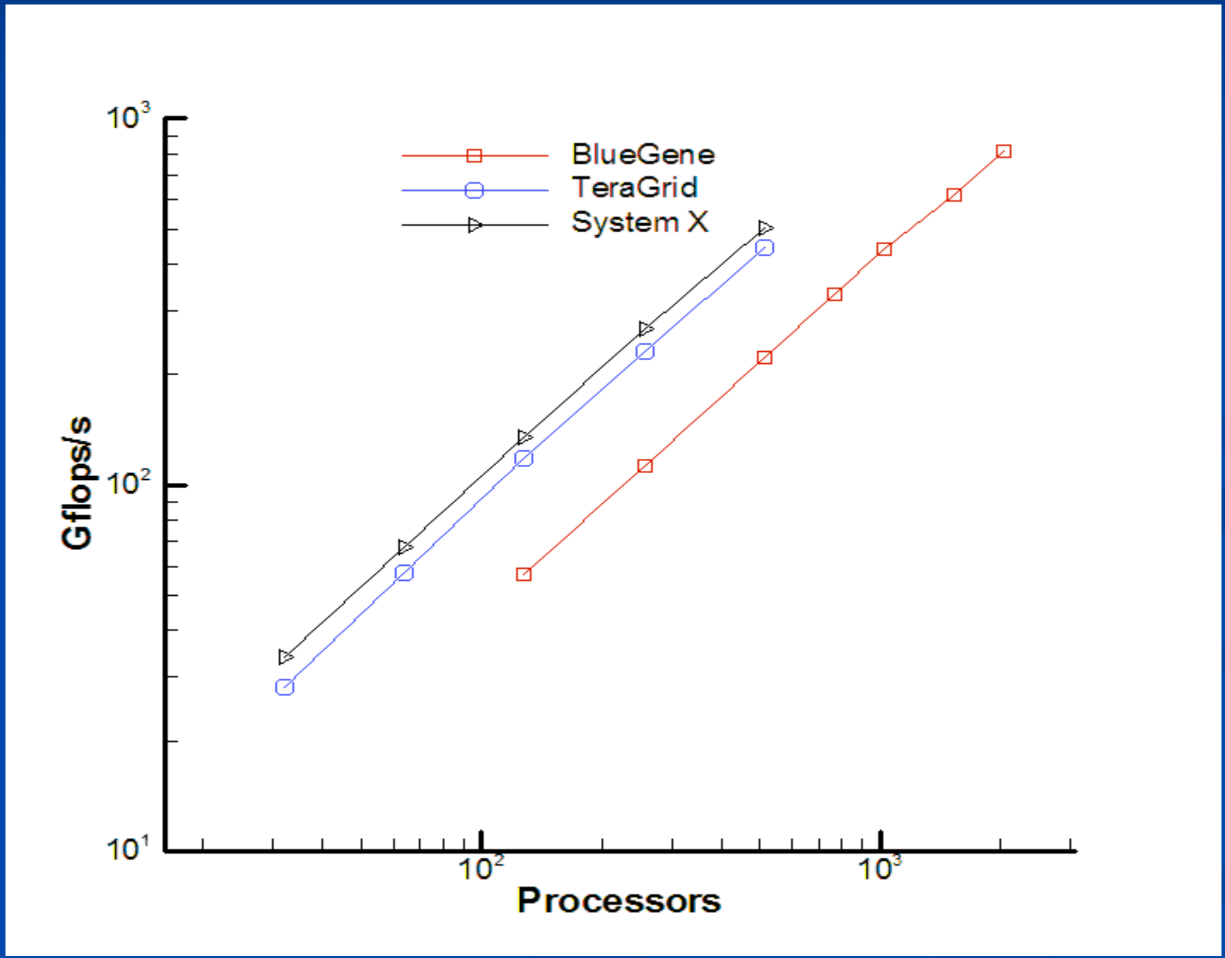
# Parallel Performance of PETSc-FUN3D

**3D Mesh: 2,761,774 Vertices and 18,945,809 Edges**

**TeraGrid: Dual 1.5 GHz Intel Madison Processors with 4 MB L2 Cache**

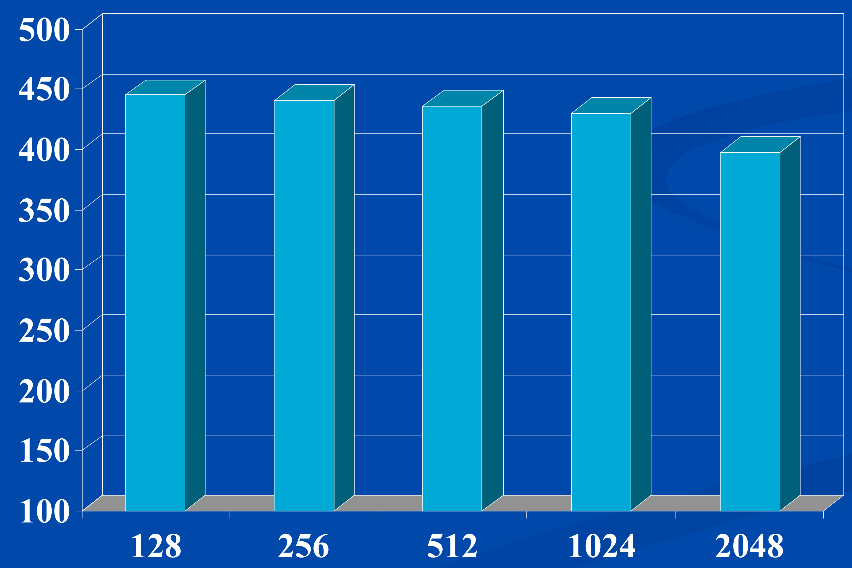
**BlueGene: Dual 700 MHz IBM Processors with 4 MB L3 Cache**

**System X: Dual 2.3 GHz PowerPC 970FX processors with 0.5 MB L2 Cache**



# BlueGene Per-Processor Performance

- Insignificant loss in performance due to parallelism even for strong scaling
  - 16% of peak on 128 processor vs. 14% on 2048 processors
  - Machine mode changes from coprocessor to virtual node
- In the overall parallel performance, poor per-processor part is the real “culprit” and not the scalability



# Conclusions

# Designing Parallel Programs

- Common theme – think about the "global" object, then see how MPI can help you
  - Solve a bigger problem
  - Cut down the execution time
- Also specify the largest amount of communication or I/O between "synchronization points"
  - Computation to communication ratio
  - Collective and noncontiguous I/O
  - Point to point vs. RMA

# MPI

- MPI is a proven, effective, portable parallel programming model
- MPI has succeeded because
  - rich features
  - control on data placement (critical for performance)
  - complex programs are no harder than easy ones
  - open process for defining MPI led to a solid design

# PETSc Library

- PETSc provides scalable linear and nonlinear solvers
  - convenient algorithmic experimentation
  - portable wherever MPI is available
  - used in a variety of application areas
- From a performance standpoint, parallel programming is *easy* but sequential programming is *difficult*!

# Acknowledgements


- MPICH Team at MCS (Bill Gropp, Rusty Lusk, and Rajeev Thakur in particular)
- PETSc Team and David Keyes
- LCRC Team (Susan Coghlan, John Valdev, and Ray Bair)
- Computer time was provided by ANL for Jazz, SDSC for TeraGrid, and Virginia Tech for System X